Floating Point Functional Cores for Reconfigurable Computing Systems

Clay Gloster, Jr.
Howard University
Dept. of Electrical & Computer Engineering
2300 Sixth Street, NW
Washington, DC 20059

Abstract— **Traditionally, RC system designers were required to transform applications written using floating point arithmetic into functionally equivalent applications that use integer arithmetic. This was due to the limited logic capacity of FPGA devices. Recent advances in the logic capacity (> 2 million gates) and the maximum clock rate (> 100MHz) for FPGA devices has enabled the use of floating point modules in reconfigurable computing systems.**

This paper presents a library of pipelined floating point functional units, called functional cores, that can be used in isolation or with a standard reconfigurable instruction set computer. A standard interface allows the simple interconnection of these cores to form more complex functional units. The use of standard single precision floating point units in reconfigurable computing system design also facilitates hardware debugging when implementing algorithms that were previously written in software using floating point arithmetic and removes the need for extensive precision analysis. The functional cores presented include: an adder, subtractor, multiplier, divider, accumulator, multiply-accumulate unit, complex arithmetic multiplier, a complex arithmetic accumulator, a core specifically design for LU decomposition, and a core used to compute the Discrete Fourier Transform. The paper concludes with area and performance data for these functional cores mapped onto a typical commercial FPGA device.

I. Introduction

Reconfigurable computing (RC) has emerged as a viable computing option for applications that require high performance. RC systems are a combination of hardware/software data processing platforms that implement computationally intensive algorithms in Field Programmable Gate Array (FPGA) hardware devices. Typical RC systems yield 10X to 100X improvement in processing speed over conventional CPU-based "software-only" systems. Reconfigurable computing systems are also "in-circuit" re-programmable, allowing data collection or processing configurations to be changed in microseconds. These RC systems combine the flexibility of general-purpose processors with the speed of application specific processors. By mapping an application to an RC system, the system designer can personalize the hardware for each particular application, thereby increasing system performance without sacrificing system flexibility.

In the past, system designers have generally had two options for implementing their system: 1) design a system that is flexible by writing software that executes on a general purpose processor (desktop computer) or 2) design an application specific integrated circuit (ASIC) for a particular algorithm [1]. The first option yields systems that are capable of performing many common functions in an acceptable amount of time. Because the design of the general purpose processor is not centered around a single function, or small group of functions, it can be programmed to perform a large variety of tasks. Unfortunately, this flexibility may result in poor performance for specialized tasks. The second option often results in the greatest system performance gains, but this comes at the expense of system development time and chip fabrication costs that are significantly higher than the cost of a desktop computer.

This problem has led system designers to attempt to find methods for improving the processing time of applications executed on general purpose machines while minimizing the associated development costs. Early attempts to improve computer system performance were made in the mid 1970's and centered around adding many complex instructions to the computer's instruction set. The use of these complex instructions reduced the number of instruction fetches and decodes required for typical programs, thus reducing the number of main memory accesses [2]. This led to the introduction of the Complex Instruction-Set Computer (CISC). Since it is very difficult for compilers to recognize these complex instructions in traditional high-level language programs, only a small subset of the available instructions were actually used in typical programs [3]. As a result, in the 1980's, computer architects focused on reducing the number of instructions available [4]. This led to the introduction of the Reduced Instruction Set Computer (RISC). This reduction in the number of available instructions allows for a faster clock rate, that typically leads to a reduction in execution time.

The implementation of coprocessors, or more generally, special purpose processors, is an attractive alternative to the desktop computer and the ASIC for system design. Coprocessors and special purpose systems are developed to perform a small number of functions very efficiently [4]. Because they are designed in this manner, they are generally very fast at a certain type of application, but incapable of performing a wide variety of applications.

This paper utilizes a reconfigurable coprocessor that is specifically tailored for each application. It presents a solution that merges the advantages of typical general purpose processors and ASICs. Similar to a general purpose processor, the reconfigurable processor can be configured to execute a wide variety of tasks. Similar to an ASIC, the reconfigurable processor is designed specifically for each application resulting in increased performance.

The paper also presents a library of pipelined floating

This work was supported under NASA contract number #AIST-0016-0044.

point functional units, called functional cores, that can be used in isolation or in the reconfigurable processor. Not only does the use of these units facilitate system development and debugging, but also these units utilize a small percentage of the available FPGA logic capacity resources.

II. A Reconfigurable Processor

The concept of developing a reconfigurable processor is not new, in fact, several reconfigurable processors have appeared in the literature over the past 10-15 years [5], [6]. These approaches utilize the FPGA fabric to implement a processor with a small instruction set. Since these processors run at clock speeds that are much less than typical microprocessors, running all applications on such a processor would result in significantly slower execution times. However, since the FPGA can be loaded with a new processor, the *best* reconfigurable processor for the target application can be used resulting in execution times that can be significantly faster than typical general purpose processors.

In this paper, a standard data unit and a standard control unit are used as the infrastructure to design a large library of reconfigurable processors. In contrast to the approach of developing a detailed FPGA-based design for each application, our approach is to combine pre-designed units with functional cores to create many unique reconfigurable processors. Ultimately the synthesis of future reconfigurable processors can be fully automated by developing a program that uses a hardware description language, i.e. VHDL, to stitch together all required processor components.

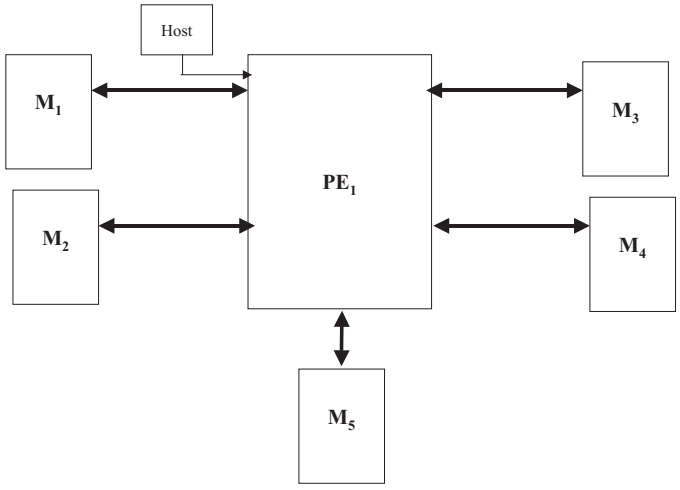

Fig. 1. A reconfigurable processor with 5 local memories.

Fig. 1 presents a typical reconfigurable processor architecture with five local memories. With a large number of memories, the processor can be designed to fetch multiple input operands from different memories while simultaneously writing a result to another memory. For example, given five memories, up to four inputs can be loaded at a time while writing an output to the fifth memory. This could provide a 4X speedup over systems that contain only a single memory, i.e. general purpose processors.

Fig. 2 shows a single reconfigurable processor, also called

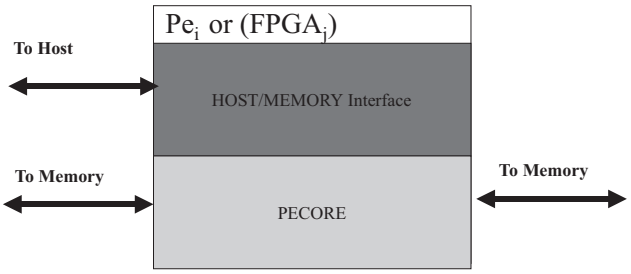

Fig. 2. Processing element architecture.

a processing element (PE). Each PE is interfaced with several memories: the host memory and the FPGA's local memory. A host memory interface manages the transfer of data between the host memory and the FPGA's local memory. For applications that require large amounts of data to be transferred from the host memory to the FPGA's local memory, this could present a significant performance bottleneck. However, future RC systems that allow the FPGA and the host to share a memory can alleviate the need to transfer data between the host and FPGA memory along with the removal of the performance bottleneck.

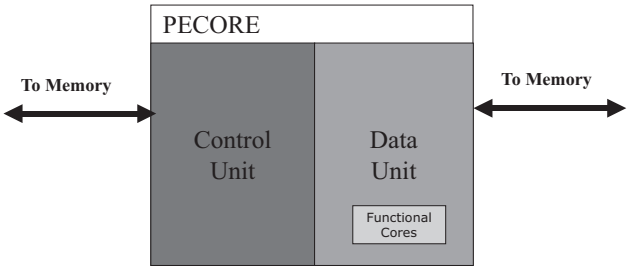

Fig. 3. Processing element core control and data units.

Each PECore, shown in Fig. 3, contains a control unit and a data unit. The control unit is a finite state machine that manages memory read/write transactions, initiates the instruction fetch-decode-execution cycle of the processor, and determines when instruction processing is complete. Once processing is complete, the control unit signals the host to turn control back over to the host/memory interface. The data unit contains a register file, a program counter, an instruction register, several memory address registers, counters for determining when vector instructions are complete, and one or more functional cores that are application specific. All registers and counters are 32-bits in length. The current implementation of the reconfigurable processor contains 8 registers, 5 memory address registers, and a maximum of 7 functional cores.

Our reconfigurable processor allows the reuse of particular op-codes for different instructions that are loaded into the FPGA. Details of the assembly language and instruction set opcodes are found in [7], [8]. This combination reconfigurable processor and reconfigurable instruction set provides the benefits of both RISC and CISC processors. It offers the performance of a RISC processor because the maximum number of instructions is limited, i.e. 32 instructions. This fixes the decode logic, the critical path, and ultimately the maximum clock period of the processor.

It provides the benefits of a CISC processor since we can potentially load an infinite number of processors into the FPGA, each containing a unique instruction set composed of a limited number of instructions, i.e. 32 instructions. An assembler reads assembly language programs(called session files) written using our instruction set and executes them directly on a reconfigurable computer.

Each new instruction set is obtained by inserting a new functional core into a standard data unit and connecting it to a standard control unit. While the number of unique control units and data units contained in the library is small, there are theoretically an infinite number of functional cores. Hence, given an FPGA with infinite logic capacity, one can build a library that consists of an infinite number of reconfigurable processors, each containing a limited number of instructions, i.e. 32.

A novel feature of the proposed reconfigurable processor is that a large percentage of the architecture is fixed for a given commercial reconfigurable computing system. While the host memory interface is unique for each RC system, it only needs to be designed once for each board and can be reused for each new reconfigurable processor that is developed. This is also true for the standard control unit and data unit. Only slight modifications need to be made to these modules that we have modeled using the VHDL hardware description language to produce a new reconfigurable processor that can be tailored for a single or a small set of applications. In most cases, only the application-specific functional cores need to be developed by the designer for each new application. We utilize a VHDL module generation program (FunCoreGen), developed at Howard University, to produce a VHDL description of complex functional cores given a simple netlist description of the complex core.

III. Floating Point Functional Cores

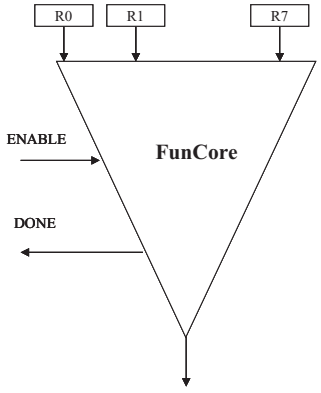

Fig. 4. Function core definition (Type I).

A functional core is a floating point unit that contains n inputs and a single output. An 8-input functional core is shown in Fig. 4. This core must be incorporated into a reconfigurable processor with a register file containing a minimum of 8 registers, since the outputs of each register are connected to the inputs of the function core.

There are currently two types of functional cores supported by the standard control unit: cores *WITHOUT* an accumulator (Type I) and cores *WITH* an accumulator (Type II). All function cores (Types I and II) have simple control, including an *ENABLE* pin, used as a signal that valid data is available on the core inputs, and a *DONE* pin, used to signal that the core output data is valid. In the case of multiple functional cores contained in a single data unit, all register outputs are shared by each function core.

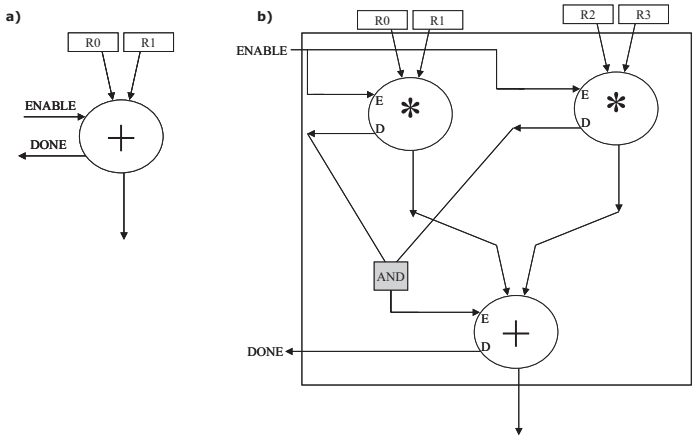

Fig. 5. Example function cores (Type I).

Fig. 5 presents two sample function core, both of Type I. Fig. 5a is a simple two input functional core that consists of a single pipelined floating point adder. Complex functional cores are built by combining simple cores available in our library as shown in Fig. 5b. When creating complex functional cores, the *DONE* signals of the initial levels of the tree are connected to the *ENABLE* signals of the subsequent level of the balanced tree. Hence the functional core of Fig. 5b can be used to compute $Y = (R0 \times R1) + (R2 \times R3)$. In order for the complex core to be used with the standard control unit, it must contain a single *ENABLE* input pin and a single *DONE* output pin.

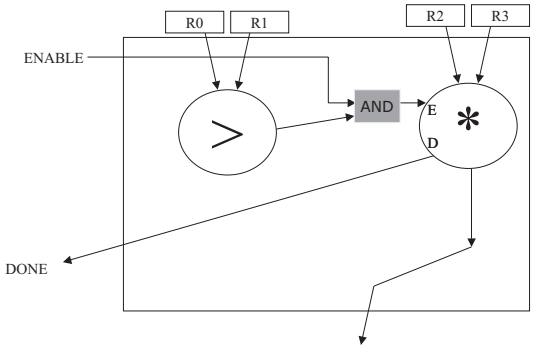

Fig. 6. A sample functional core that includes a conditional.

Type I cores can also include conditionals as shown in Fig. 6. This same core computes the expression $Y = R2 * R3$ if $R0 > R1$. The else conditions can be incorporated using multiplexors whose output are selected by the output of the floating point comparison units. Once

the designer has built a greater than, less than, and equal units to unit, along with multiplexors, a large number of conditional units can be developed.

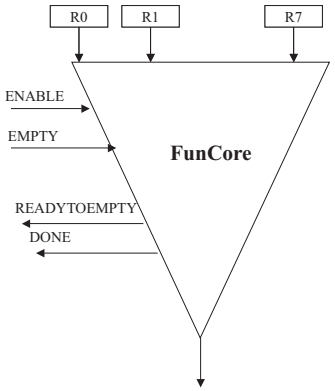

Fig. 7. Functional core definition with an accumulator (Type II).

Type II cores, shown in Fig. 7, with an accumulator at the output of the core have two additional pins: *READYTOEMPTY* and *empty*. The *READYTOEMPTY* pin signals that the accumulator in the unit can be emptied at this time. To empty the accumulator, the *empty* input pin is asserted. Since the accumulator contains a pipelined floating point adder, it takes many clock cycles to finish summing up the partial sums producing a single result. Once the emptying process is complete, the *DONE* signal is asserted signaling that the core contains valid data on it's output.

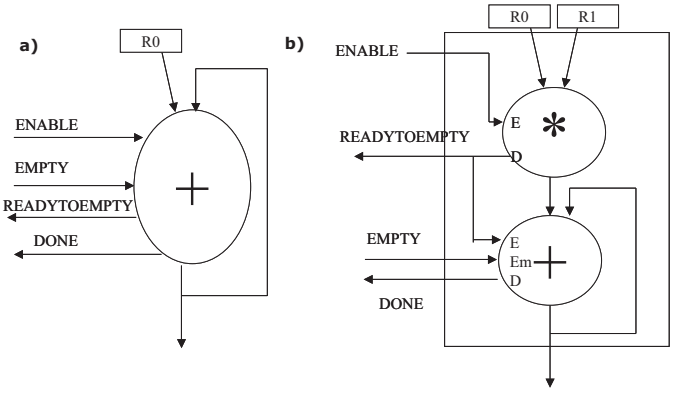

Fig. 8. Example function cores with an accumulator (Type II).

Sample Type II cores are shown in Fig. 8. All of these units must contain an accumulator at the output. The first unit is a simple accumulator module that was built using a special floating point adder that holds the data in the pipeline until it receives valid data on it's input or until it is emptied. It contains, a special input selector that recirculates the adder output back into one of it's inputs along with eight registers used to store partial sums that need to be added together. A simple finite state machine manages accumulator operation. To operate a Type II core, the *ENABLE* signal is asserted each time valid data is available on it's input. The empty signal is then asserted after all input data has been read into the unit and a READYTOEMPTY=1 has been observed. Fig. 8b shows a multiply-accumulate module that was built by connecting a floating point multiplier to the floating point accumulator module. While the core in Fig. 8a is always ready to be emptied, the core in Fig. 8b is only ready to be emptied when data exits the multiplier.

All floating point cores developed at Howard University have a latency of either 8 or 16 cycles. The latency was standardized to facilitate the insertion of delays when connecting these units together to form more complex balanced trees. Since the latency is a multiple of 8, only two delay unit types needed to be included in the library (Delay8 and Delay16). Using non-standard latencies would introduce additional complexity in delay insertion to ensure that the resulting trees of interconnected floating point units are balanced.

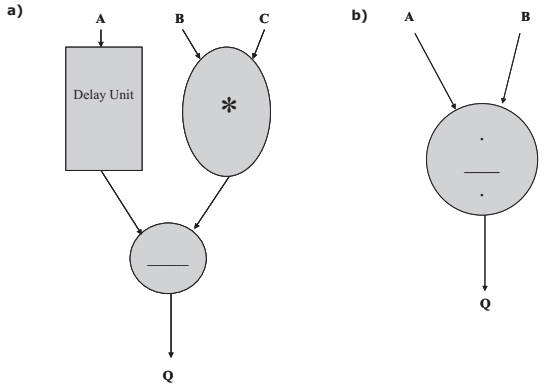

Fig. 9. Two functional cores used for LU decomposition.

Fig. 9 shows the complex cores used to build a reconfigurable processor that implements LU Decomposition. The LU decomposition algorithm can be used to solve a system of simultaneous linear equations without performing matrix inversion. The problem of LU decomposition can be stated as follows: given a matrix **A**, find two matrices **L** and **U** such that **LU** = **A**, where **L** is a lower triangular matrix, and **A** is an upper triangular matrix. Details of the LU decomposition algorithm can be found in [7].

For LU decomposition we use a reconfigurable processor with two functional cores. One functional core is used to perform all of the vector division operations while the other performs multiplication and subtraction. Since the multiplier has a latency of 8, as is the case for all simple Type I cores except the divider (latency = 16), an eight stage delay unit must be used. With these two cores, we achieved a speedup up 7X over running the same algorithm on a general purpose processor [7].

The most complex core that we currently have in our library is used to compute the 1024-point Discrete Fourier Transform (DFT) . It can accommodate both real and imaginary input data and can compute both the DFT and it's inverse. This core contains a theta unit to compute the value of the angle for the DFT, a sine/cosine unit look-up-table, a complex multiplier, and a complex accumulator. Since we needed to develop the complex multiplier and accumulator for the DFT, we also included these units in the floating point library. Models for all functional cores written using the VHDL language can be found in [9].

IV. Floating Point Area/Performance Statistics

In our experiments, we synthesized all floating point function cores targeting an FPGA board, available at Howard University, containing an XCV2000E FPGA with approximately 2 million system gate equivalents. This board contains four, 8MB local FPGA memories and one, 4-MB local FPGA memory. The FPGA board has a maximum clock frequency of 100 MHz.

The entire library was modeled in VHDL and simulated extensively. The resulting models were mapped onto the FPGA using commercial logic synthesis and FPGA placement and routing tools. The goal of this experiment was to investigate the feasibility of using floating point units for various applications in an RC system.

TABLE I
Functional core area/performance

Core Name	# of LUTs/ (Utilization)	Clock Frequency (MHz)
Adder/Subtractor	526 (1%)	63.9
Accumulator	806 (2%)	46.4
Complex Multiplier	1025 (2%)	67.0
Multiplier	1245 (3%)	64.7
Complex Accumulator	1610 (4%)	43.9
Multiply-Accumulate	2158 (5%)	46.4
Divider	2072 (5%)	56.3
LUCore	3868 (10%)	56.3
DFTCore	6094 (15%)	39.1

Table I presents the results obtained in our experiment. The table reports the name of the floating point functional core, the number of lookup tables (LUTs) required for the core, the FPGA utilization as a percentage, and the maximum clock frequency that the core can accommodate. Results demonstrate that approximately 100 floating point adders could be placed on a single XCV2000E part. This is an interesting result as in the past only 5-10 adders could fit on a typical FPGA. Since the floating point multiplier requires about 3% of the part, a total of approximately 30 multipliers can be placed on a single FPGA. Hence functional cores composed of many adders and multipliers are feasible. Even the most complicated functional core used for calculating the DFT can be replicated approximately 6 times on the XCV2000E FPGA. The maximum clock rates for all functional cores ranges from 39.1 MHz for the DFT core to 67.0 MHz for the complex multiplier.

This paper has presented a reconfigurable processor architecture with a very flexible datapath containing unique functional cores that are tailored for each application. A novel instruction set that allows op-code re-use has been successfully tested. Results demonstrate that, floating point units *can* be used on current FPGAs in spite of the design complexity as compared to integer units. The benefits of using floating point units are:

1. Applications requiring a large dynamic range can be developed fairly simply.
2. Hardware debugging is simplified when given an initial application that uses floating point numbers since there is a one-to-one correspondence between data outputs of the hardware and software.
3. Using floating point functional cores in a reconfigurable processor on an FPGA can result in significant performance gains over software executing on a typical general purpose processor.

ACKNOWLEDGMENT

The acknowledges Thomas Flatley, of NASA GSFC, for allowing us access to the facilities located in the Ground Systems Hardware Development Laboratory at NASA Goddard Space Flight Center. The project also thanks Esther Dickens and Hassan Phillips, of Howard University, for their help with the VHDL modeling and simulation of the LU decomposition cores. Tiffany Crawford, now attending UC Berkeley, was instrumental in the completion of the floating point divider core that was presented in this paper.

References

[1] J. Villasenor and W. H. Mangione-Smith. Configurable Computing. *Scientific American*, June 1997.

[2] A. K. Agerwala and T. G. Rauscher. *Foundations of Microprogramming Architecture, Software, and Applications.* Academic Press, New York, N.Y., 1976.

[3] D. A. Patterson and D. R. Ditzel. The Case for RISC. *Computer Architecture News*, 8(6):25–33, October 1980.

[4] D. Wo and K. Forward. Compiling to the Gate Level for a Reconfigurable Co-Processor. In *1985 International Symposium On Circuits and Systems*, pages 2–4, 1985.

[5] M. J. Wirthlin and B. L. Hutchings. DISC: A Dynamic Instruction Set Computer. In *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[6] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. Splash: A Reconfigurable Linear Logic Array. In *International Conference on Parallel Processing*, pages 526–532, 1990.

[7] C. Gloster and H. Phillips. A reconfigurable instruction set processor and assembler. *NASA Earth Science Technology Conference*, June 2002.

[8] C. Gloster and E. Dickens. A reconfigurable instruction set microcomputer. *MAPLD International Conference*, September 2002.

[9] http://www.imappl.org/čgloster/rare, May 2003.